Divo: Learning a Stable and Effective Query Optimizer with a Diverse Workload

TIANYI CHEN, Key Laboratory of High Confidence Software Technologies, CS, Peking University, China
JUN GAO*, Key Laboratory of High Confidence Software Technologies, CS, Peking University, China
YAOFENG TU*, ZTE Corporation, China
YANG LIN, ZTE Corporation, China
MO XU, ZTE Corporation, China
YINJUN HAN, ZTE Corporation, China

Learned query optimizers (LQOs) have shown remarkable progress in recent years. Despite achieving competitive performance compared with traditional methods, current LQOs struggle to retain stability and generate efficient plans when exposed to workloads with substantial diversity, presenting a major challenge in real-world deployment. Divo is a learned query optimizer designed to overcome performance degradation observed in training on diverse workloads. First, Divo proposes a template-evolving query generator to provide diverse queries for sufficient training. By modifying and combining query templates, the query generator synthesizes 3,000 informative new queries with fidelity to existing workloads. Second, Divo establishes a two-phase model training pipeline to enhance RL training with numerous plans collected in advance. We collected over 100,000 plans from public workloads and diverse generated queries to form static experiences, which are effectively leveraged on various training workload configurations to enhance Divo's plan generation through augmenting auxiliary components. Third, Divo proposes a diversity-aware loss function to learn from diversified input queries stably. By learning to predict probability distributions of latencies, Divo flexibly handles various queries and tolerates inconsistent ground truth latencies from training queries. We evaluate Divo's performance on PostgreSQL using a complicated mixed workload of JOB, DSB, Extended JOB, Stack, and generated queries, with three different training and testing workload configurations split by template. Experiments demonstrate Divo's advantage on total execution latency with a 1.3× speedup relative to PostgreSQL, which is 14.9× better than six existing LQOs on average.

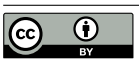
CCS Concepts: • **Information systems → Query optimization**; • **Theory of computation → Sequential decision making**.

Additional Key Words and Phrases: Learned Query Optimization, Query Generation, Diverse Query Workload, Deep Reinforcement Learning

ACM Reference Format:
Tianyi Chen, Jun Gao, Yaofeng Tu, Yang Lin, Mo Xu, and Yinjun Han. 2026. Divo: Learning a Stable and Effective Query Optimizer with a Diverse Workload. *Proc. ACM Manag. Data* 4, 1 (SIGMOD), Article 27 (February 2026), 24 pages. https://doi.org/10.1145/3786641

*Corresponding authors.

Authors' Contact Information: Tianyi Chen, tianyichen@stu.pku.edu.cn, Key Laboratory of High Confidence Software Technologies, CS, Peking University, Haidian Qu, Beijing Shi, China; Jun Gao, gaojun@pku.edu.cn, Key Laboratory of High Confidence Software Technologies, CS, Peking University, Haidian Qu, Beijing Shi, China; Yaofeng Tu, tu.yaofeng@zte.com.cn, ZTE Corporation, China; Yang Lin, lin.yang@zte.com.cn, ZTE Corporation, China; Mo Xu, xu.mo1@zte.com.cn, ZTE Corporation, China; Yinjun Han, han.yinjun@zte.com.cn, ZTE Corporation, China.



ACM 2836-6573/2026/2-ART27
https://doi.org/10.1145/3786641

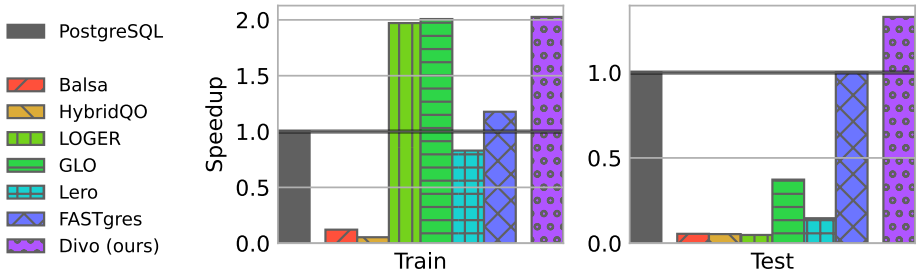


Fig. 1. Speedup of LQOs relative to PostgreSQL when trained and tested on subsets of a mixed workload.

1 Introduction

With the rapidly increasing amount of data, query execution latency has become a key bottleneck of numerous application scenarios. Query optimization aims to find an optimal execution plan with the lowest latency for each input query [25]. While traditional query optimizers of DBMSs exhibit stable performance built upon human expert knowledge, learned query optimizers (LQOs) have recently been proposed and shown competitiveness [11, 18, 40].

Based on the strategy of plan generation, LQOs can be categorized into *generator-like* and *steerer-like* algorithms [3, 45]. Generator-like LQOs, *e.g.*, Balsa [37], LOGER [2], *etc.*, directly generate plans without empirical rules. The plan generation strategy is mostly controlled by a latency predictor, which predicts the minimal observed latency for intermediate plans (subplans) and is iteratively refined by reinforcement learning (RL). In contrast, steerer-like LQOs, like Lero [46] and FASTgres [34], invoke a traditional optimizer (mostly PostgreSQL's) to generate plans and select the best one from the candidates with a plan comparator. These steerer-like LQOs are generally more stable than generator-like methods because the traditional optimizer undertakes plan generation. With sufficient training, both kinds of LQO outperform PostgreSQL's optimizer when training and testing on identical query templates.

However, existing LQOs consistently fail to maintain performance when we increase the query workload diversity to imitate general applications. Figure 1 shows the relative speedup of these LQOs to PostgreSQL on a mixed workload of JOB [12], Extended JOB [17], DSB [4], and Stack [16]. We trained these LQOs with half the templates and used the rest to test their speedup. We can see that generator-like LQOs failed to achieve even half of PostgreSQL's performance, and none outperformed 1× speedup on the unseen testing workload, showing the difficulty of LQO's learning.

As recent progress in LLM sheds light on the definite relationship between the quality and diversity of training data and model performance [10, 26, 32], improving the effectiveness of training queries and plans becomes a natural intuition. A diverse training workload is expected to equip LQOs with sufficient knowledge and lead to better performance. Despite the appealing idea, existing methods perform conversely and show limitations of current LQO training schemes, which face three major challenges.

The lack of diverse informative queries. The most frequently used workloads, including JOB [12], Stack [16], *etc.*, contain only 10 to 30 query templates with complicated join predicates. Several benchmarks, *e.g.*, DSB [4], can generate queries from templates with different filter predicates. However, these queries are seldom informative since most queries of each template share an identical optimal join order. For example, queries of DSB's template 18 share a near-optimal join order that first joins '*catalog_sales*' and '*date_dim*' and then '*item*', *etc.*, providing no more knowledge than the template itself. Therefore, generating queries of diverse templates is crucial

to LQOs. Besides, researches in various areas [6, 14, 29, 30] have demonstrated the necessity of fidelity, *i.e.*, the similarity between synthetic and real data, for performance on real user inputs. A query generator should produce diverse, informative queries with fidelity at scale, posing the need for a subtle implementation.

Obstacles to effective plan exploration and utilization. As an LQO learns from observed *plans* rather than queries to refine the query optimization policy, leveraging diverse training queries remains a key issue. Improving an LQO requires learning from efficient plans that outperform existing ones, necessitating effective exploration in an exponential search space [2]. However, a large and diverse workload challenges sufficient RL exploration and requires considerable overhead. An alternative is to collect subplans for individual queries in advance of training, thereby circumventing exploration inefficiency. However, collected observed latencies are immutable during training and become obsolete through exploration, eventually hindering efficient plan generation. These issues pose a unique challenge for LQO training.

Complicated distribution of diverse training queries. While a diverse training workload enriches knowledge for plan generation, the varying scales of latencies create inherent biases towards queries with large latency spans, which produce higher loss values. Moreover, the latency prediction task is easily affected by variable latencies with identical feature inputs. For example, JOB queries 8c and 8d have the same predicate selectivities, but 8c is about 2× slower in PostgreSQL because joining '*ci*' and '*rt*' in 8c yields 9× more data than in 8d. Subtle differences intermittently manifest among queries, inducing inconsistent ground truths that eventually destabilize RL training. A training method that flexibly handles inconsistency and intrinsic bias is thus pivotal to the issues.

To handle the aforementioned challenges, We propose Divo, an LQO established on diverse workloads. With a hybrid framework combining the advantages of generator-like and steerer-like methods, Divo generates plans with a *latency predictor* and replaces poor plans with PostgreSQL's plans with a *comparator* [3, 46]. We carefully design Divo with the following main contributions.

Template-evolving query generator. To address the diversity challenge, Divo introduces a *template-evolving* query generator to synthesize templates through join graph operations on existing workloads. Divo applies two core operations, *reduction* for node removal and *combination* for graph merging, to evolve graph structures where nodes and edges represent tables and join predicates. This approach ensures fidelity by preserving join relationships from original workloads while systematically expanding template diversity. A weighted sampling strategy further ensures diversity and fidelity with fair sampling and exponential weight decay for generated templates. We generate 3,000 queries based on JOB [12], Extended JOB [17], DSB [4], and Stack [16] to furnish Divo with sufficient knowledge.

Diverse plan-augmented RL with one-time collection. We propose a *two-phase* training pipeline for Divo to effectively collect and leverage plans from diverse generated queries. Although unsuitable for direct RL training, diverse queries can provide high-quality supervision for auxiliary components, specifically the plan comparator in Divo. First, Divo collects extensive *complete* execution plans to form *static experiences*, which retain validity with *fixed* observed latencies throughout training. In the second phase, Divo learns plan generation with value-based RL and augments the plan comparator with static experiences, thereby simultaneously improving plan generation through a shared query encoder. Learning from complete plans enhances Divo without suffering from outdated prior plan generation strategies. Moreover, static experiences can be reused across various training workload settings with a one-time collection overhead.

Diversity-aware probability distribution loss. Divo sufficiently handles the complicated distribution of diverse queries with a *diversity-aware* loss function, which integrates KL divergence rather than relying solely on MSE loss used by other LQOs [2, 3, 37]. The diversity-aware loss enables Divo

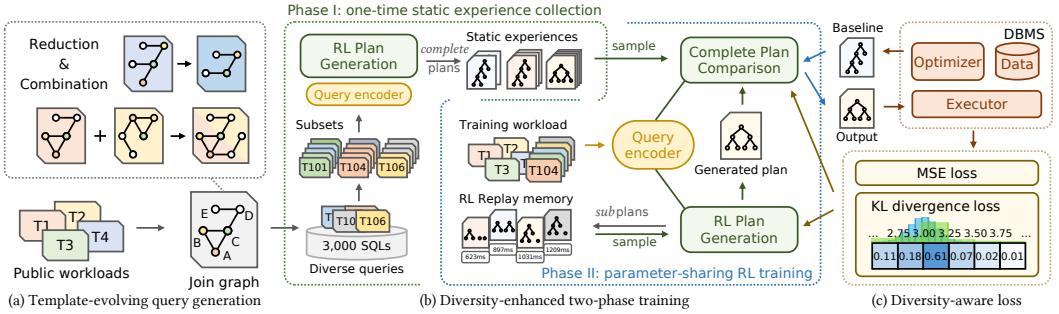


Fig. 2. An overview of Divo's framework.

to learn an additional task to predict the probability distribution of latency in different ranges. On the one hand, the diversity-aware loss naturally handles intrinsic biases and inconsistent latency values with multiple class labels of latency ranges. On the other hand, probability distributions provide a fine-grained view for the comparator. Instead of binary predictions in Lero [46], GLO [3], *etc.*, probability distributions enable Divo to calculate the mathematical expectation of speedup relative to PostgreSQL for each plan to make plausible decisions.

We tested Divo and other LQOs on multiple different splits of JOB [12], Extended JOB [17], DSB [4], Stack [16], and 100 random generated queries. Our experiments ensure a challenging setting where test set templates are unseen in the training workload. On the testing workload in Fig. 1, Divo benefits from the previously mentioned contributions and presents a 1.3× speedup, which is 14.9× better than six compared LQOs on average and indicates a stable performance on diverse queries.

2 Related Work

Generator-like and steerer-like LQOs. A generator-like LQO constructs plans by consecutively selecting join operations for subplans, mostly using a learned latency predictor to estimate the minimal latency of a subplan *observed* through training. The LQO enumerates possible join operations, predicts latencies of corresponding subplans, and selects the next subplan with the lowest latency. This procedure is repeated until all tables are joined together. With ReJOIN [18] and DQ [11] as prototypes, several generator-like LQOs including Balsa [37], HybridQO [39], LOGER [2], *etc.*, have exhibited impressive performance in multiple workload splits.

On the contrary, steerer-like LQOs invoke the underlying traditional optimizer to generate a set of candidate plans and use a comparator to select the best one. These methods diversify candidate plans with various strategies. Bao [16] disables specific join or scan operators. Lero [46] alters the cost estimations. Some steerer-like LQOs, such as FOSS [45], modify the candidate plans to improve performance. GLO [3] borrows the idea and uses a comparator to select between generated plans and PostgreSQL's plans.

Synthetic query generation. Effective ML-based database components require substantial training queries, as demonstrated in domains like cardinality estimation [9]. Early approaches like RAGS [27] and SQLSmith generate queries randomly but cannot produce diverse join predicates essential for LQO training. Template-based methods, such as TPC [22] and DSB [4], produce queries with a fixed set of variants through parameterized templates. Several methods [1, 19] further apply constraints to template-based methods to improve informativeness, but queries from the same template typically share the optimal plan and cannot provide diverse knowledge. Recent learning-based

methods improve query generation for various requirements. TreeGAN [15] addresses fidelity to real queries, and LearnedSQLGen [42] enforces constraints with RL to generate informative queries. However, these methods require additional training overhead on the underlying schema and primarily yield queries of low complexity (fewer than 3 joins). Our template-evolving query generator addresses these limitations by systematically evolving existing templates to produce diverse, informative queries with fidelity at scale.

Deep reinforcement learning. Deep RL is a technique frequently used in learned database components [13, 28, 35]. Divo follows the value-based RL paradigm, which estimates the expected returns of state-action pairs to obtain a policy for the current task [31]. Most generator-like LQOs, including Neo [17], Balsa [37], GLO [3], *etc.*, use a modified implementation of Deep Q-learning (DQN) [20]. As a pivotal value-based RL algorithm, DQN establishes an *experience replay memory* $\mathcal{M}$ to enhance data efficiency and diminish correlations between samples. Instead of adopting the target network in DQN, generator-like LQOs establish a lookup table of the minimal observed latencies for subplans. Divo follows this implementation. More recently, Farebrother et al. [7] have shown the advantages of training the value model with classification tasks on performance and stability with substantial experiments. Divo adopts the idea and proposes the *diversity-aware* loss of latency distribution.

3 System Overview

Figure 2 demonstrates Divo's high-level architecture and workflow. This section provides a holistic view of (1) model components adopted from prior work, (2) the training pipeline, and (3) the inference pipeline. Implementation details of Divo's main contributions are explained in §4-5.

3.1 Components of model architecture

Divo's model architecture consists of three components: the query encoder, the latency predictor, and the plan comparator. We briefly explain the implementation of these components as follows.

Query encoder. The query encoder takes queries q as input and provides table embeddings for the downstream latency prediction and comparison tasks. We adopt the query encoder in GLO [3], which embeds the information of q and the statistics from the database into a graph $\mathcal{G} = (V, E)$. The query encoder uses a 2-layer Graphormer [38] model to output vertex embeddings e_t, where $t \in V$ is a table. We denote the encoder as F_{enc} below.

$$F_{\text{enc}}(q) = \{e_t \mid t \in V \wedge t \text{ is a table}\}$$

To embed table and column information into q, Divo establishes $\mathcal{G}$ as a join graph of table and column vertices, where edges represent column filters or join predicates. We adopt the representation vectors used in GLO and further add edge features to enrich the information. The vectors are transformed into the same feature domain with different MLP layers to feed into Graphormer.

- Table representation v_t contains (1) estimated row count, (2) estimated total selectivity (from PostgreSQL), (3) number of columns, and (4) number of foreign keys.
- Column representation v_c contains (1) column type, (2) distinct value proportion, (3) null value proportion, (4) existence of indexes, (5) estimated filter selectivity, and (6) existence of join predicates.
- Edge representation v_e contains (1) estimated join predicate selectivity, and (2) estimated cost.

Latency predictor. We adopt QueryFormer [44] to handle the latency prediction task. Divo represents subplans p with vertex embeddings and aggregates information into an output embedding vector $e_{\text{lat}}(p)$ with a 4-layer QueryFormer. Specifically, a subplan is represented as a forest of plan trees, where leaf nodes are tables represented with vertex embeddings e_t, and branch nodes are

joined results embedded with cost estimations from PostgreSQL's cost estimator. Divo sends the output embedding vector to an MLP M_{lat} to obtain the prediction $F_{\text{lat}}(p)$ of p's minimal observed latency.

$$e_{\text{lat}}(p) = \text{QueryFormer}^{(1)}\ (p, F_{\text{enc}}(q))$$
$$F_{\text{lat}}(p) = M_{\text{lat}}(e_{\text{lat}}(p))$$

Plan comparator. Similar to the latency predictor, the plan comparator [3, 46] uses a QueryFormer to obtain the embedding $e_{\text{cmp}}(p)$ for each plan p. The comparator takes a plan p from Divo and the default plan p_b from PostgreSQL's optimizer as input and outputs a label $F_{\text{cmp}}(p, p_b)$, which is less than 0 only when p performs better.

$$e_{\text{cmp}}(p) = \text{QueryFormer}^{(2)}\ (p, F_{\text{enc}}(q))$$
$$F_{\text{cmp}}(p, p_b) = f_{\text{cmp}}(e_{\text{cmp}}(p), e_{\text{cmp}}(p_b))$$

We will further explain the detail of plan comparison in §5.3.

3.2 Training pipeline

Divo generates templates, subsequently synthesizes queries based on public workloads, *i.e.*, JOB [12], DSB [4], *etc.*, and learns with a two-phase pipeline: experience collection and RL training. In Phase I, Divo collects numerous execution plans from these queries. The collected plans are taken as *static experiences*, which are utilized in Phase II to enhance Divo's performance.

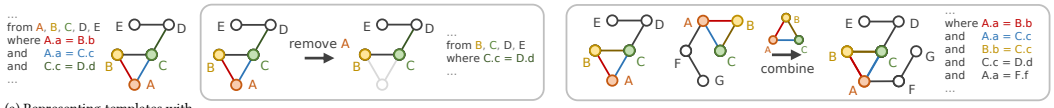
Template-evolving query generation (§4): As shown in Fig. 2(a), we propose the *template-evolving* query generator to generate diverse queries for sufficient training. The query generator produces new queries by modifying templates from the input workloads. In the generation process, Divo takes input templates as join graphs, and the generator uses two operations, (1) join graph *reduction* and (2) join graph *combination*, to randomly modify join graphs and obtain queries distinct from existing templates. We generate 3,000 queries of 1,800 different templates in our experiments. These queries are also leveraged to form a testing workload to study Divo's performance in our experiments.

Static experience collection (§5.1). We collect a total of 132,000 plans on public workloads and the generated queries in Phase I to form a set of *static experiences*, denoted as $\mathcal{S}$. To accomplish the collection, Divo searches for plans by temporarily reusing the latency predictor, which we train on different subsets of 100 queries to collect plans through RL exploration. To prevent contamination from partial or outdated knowledge and improve exploration sufficiency, we reset Divo's parameters after each collection cycle [5].

Parameter-sharing model training (§5.2). In Phase II, we train Divo with a value-based RL paradigm, which is further enhanced by static experiences. We show the training process in Fig. 2(b). For each query in the workload $\mathcal{W}$, we generate a plan p following a typical RL search pipeline. When the output plan is obtained, Divo collects the execution latency of the generated plan and updates the minimal observed latency $\hat{T}(p_t)$ for each subplan p_t. The subplans are then added to the replay memory $\mathcal{M}$ for RL training.

After the plan generation process, Divo trains the latency predictor F_{lat} and the comparator F_{cmp} with batches sampled from both $\mathcal{M}$ and static experiences $\mathcal{S}$. Divo leverages the diverse static experiences to train the plan comparator and thereby learns informative embeddings for the encoder, eventually improving Divo's plan generation and leading to better performance.

Diversity-aware loss of latency distribution (§5.3). We design a multi-objective training scheme for Divo to sufficiently fit the complicated latency value distribution of diverse training queries. As shown in Fig. 2(c), Divo trains the latency predictor with a diversity-aware loss



(a) Representing templates with join graphs (b) Join graph reduction (c) Join graph combination

Fig. 3. Illustrations of the template-evolving query generator.

function to learn from two different tasks: the regular latency prediction task identical to previous LQOs, and an additional task of predicting the probability distribution of latencies for subplans. Divo addresses the latter with a KL divergence loss, which enables Divo to effectively handle divergent latency ranges and inconsistent ground truths. In addition, the plan comparator leverages the probability distribution to obtain a fine-grained comparison between different plans.

3.3 Inference pipeline

Divo uses a typical RL search pipeline to generate plans independently in a generator-like manner, and then replaces poor plans using a comparator if needed.

Plan generation. Divo generates plans by selecting subplans. For each input query q, Divo establishes an initial subplan p_0 where all tables remain unjoined. In the t-th step of plan generation, Divo enumerates possible join actions for the current subplan p_{t-1} and obtains a set of candidates $P_t = \{\hat{p}_t^{(1)}, \hat{p}_t^{(2)}, \ldots\}$. Divo uses the latency predictor $F_{\text{lat}}(\cdot)$ to predict the minimal observed latencies of all candidate subplans. The subplan with the lowest predicted latency is then selected from P_t as the next subplan p_t. This process is repeated until a complete plan p is generated.

Erasing poor plans with comparison. Divo uses the plan comparator $F_{\text{cmp}}(\cdot)$ to recognize and erase poor plans. When the plan p is obtained, Divo invokes a traditional optimizer (PostgreSQL's in our settings) to obtain a default plan p_b. Divo evaluates the latency of p relative to the default plan p_b with the comparator and selects the better plan $p' \in \{p, p_b\}$ as the final output during inference.

We will introduce the previously mentioned components in detail in the following sections.

4 Template-evolving Diverse Query Generation

In this section, we present the basic idea of Divo's query generator, detail the main operations of the generation pipeline, and analyze the advantages of Divo's design.

4.1 Basic idea of query generation

The performance of a learned database component is critically dependent on its training data [9, 23]. Effective LQO training requires a diverse set of queries, which equip the model with sufficient knowledge to handle various future inputs. However, existing query templates offer limited variations of queries, posing the need for diverse synthetic query generation at scale.

We expect the synthetic queries generated for LQO training to satisfy several requirements. First, generated queries should exhibit substantial *diversity*. As demonstrated across ML domains, model stability and generalization require large, diverse datasets [10, 32, 36]. Second, generated queries should be *informative* for the training process. Specifically, the queries must yield non-empty results. As the DBMS executor halts on empty intermediate results, learning from such cases reinforces shortcuts or corner-case plans rather than general strategies. Moreover, generated queries should avoid Cartesian products, which are inherently inefficient and seldom occur in practice. Third, generated queries should ensure *fidelity*, *i.e.*, resemblance to existing workloads, which are representative of real-world scenarios. Fidelity helps maintain performance on real user

inputs and is essential for synthetic data and queries [21, 24]. Finally, the generation process must be *scalable*, *i.e.*, capable of rapidly producing diverse queries with low time complexity.

Note that the notion of query diversity differs from that in other domains. Even a minor modification, such as adding or removing a table from a query, creates a different execution plan and provides valuable knowledge for training. As the number of possible table combinations grows exponentially, such a straightforward approach efficiently generates a sufficient variety of new templates.

Therefore, our approach evolves templates from widely adopted workloads, including JOB [12], DSB [4], *etc.*, through rule-based operations. By appropriately removing tables or merging queries that share common join relationships, Divo ensures a certain level of fidelity by preserving authentic join predicates while offering superior scalability compared to ML-based alternatives, which require an extra training overhead.

4.2 Evolving templates with graph operations

In the following, we model query templates as join graphs, introduce two key graph transformation operations, and present the query generation pipeline with a weighted sampling strategy for template evolution.

Graph representation for templates. A template is a prototype of queries that contains unset predicate values. We represent each template as a join graph $\mathcal{G} = (V, E)$, where the vertex set V contains all tables, and the edge set E describes the join predicates. Figure 3(a) shows a query with tables A to E in the vertex set V. An edge (A, B) exists in the edge set E because a join predicate A.x = B.y presents in the template. The join graph structure determines the search space of valid plans. Specifically, two tables $\mathrm{A}, \mathrm{B} \in V$ can be joined without the Cartesian product only when a join predicate exists, *i.e.*, E contains an edge (A, B). With a correspondence between queries and join graphs, transformations on join graphs yield syntactically valid query expressions and are promising to provide diverse knowledge for LQO training.

Join graph reduction and combination. Divo use two operations to shuffle the graph structure of templates in an existing workload $\mathcal{W} = \{\mathcal{G}_1, \mathcal{G}_2, \ldots, \mathcal{G}_n\}$ and produce diverse join graphs.

- **Reduction**: given a join graph $\mathcal{G} = (V, E)$, *reduction* randomly erases vertices from the graph to form $\mathcal{G}' \subset \mathcal{G}$.
- **Combination**: given two join graphs $\mathcal{G}_1$ and $\mathcal{G}_2$ that contain a common subgraph, *combination* merges the two join graphs on vertices of identical tables to form $\mathcal{G}' = \mathcal{G}_1 \cup \mathcal{G}_2$.

Join graph reduction and combination generate new join graphs with optimal execution plans different from those of the original templates. Figure 3(b) and 3(c) demonstrate two examples for reduction and combination, respectively. The original join graph $\mathcal{G}$ in Fig. 3(a) has a vertex set $V = \{\mathrm{A}, \mathrm{B}, \mathrm{C}, \mathrm{D}, \mathrm{E}\}$, and we assume the optimal join order as joining A and B, then C, *etc.* When we erase A from V by reduction, the original optimal plan is no longer applicable, forcing the LQO to seek a distinct plan for the generated query. Similarly, merging $\mathcal{G}$ with another join graph by combination introduces new tables and possibly different optimal plans, providing extra knowledge for LQO training.

We implement reduction as a BFS-like algorithm with a time complexity of $\mathrm{O}(|V| + |E|)$. BFS inherently guarantees the obtained subgraph's connectedness, which eliminates the need for Cartesian products. Given a join graph $\mathcal{G} = (V, E)$, Divo selects a vertex $v_1 \in V$ as the initial node and adds it to the new vertex set V'. In each step, V' is enlarged by randomly selecting a vertex from V', *e.g.*, , v_1, and adding an unvisited neighbor v_2 of v_1 into V'. Divo repeats this process until $|V'|$ reaches a threshold n_r, and attaches all edges with both endpoints in V' to the edge set E' to

form the new graph $\mathcal{G}' = (V', E')$. We randomly select n_r from range $[\lceil 2/3|V| \rceil - 1, |V| - 1]$ to avoid too much difference from $\mathcal{G}$ to $\mathcal{G}'$.

Join graph combination directly merges two join graphs $\mathcal{G}_1$ and $\mathcal{G}_2$, which are further restricted to share a *connected* maximal common subgraph to strengthen the fidelity to real inputs. We show an example where we relax the restriction. JOB [12] 1a and 7a have a disconnected maximal common subgraph that contains two tables '*title*' ('*t*') and '*info_type*' ('*it*'). While both queries retrieve movie titles in '*t*', 1a finds *movies* specified by '*it*', but 7a instead seeks *persons* specified by '*it*' that appear in the cast list of movies. Therefore, the combination of 1a and 7a is likely not present in real scenarios. In contrast, 1a and 2a can be safely combined since they share a connected maximal common subgraph of '*title*' and '*movie_companies*' and both refer to the titles and companies of specific movies. In addition, we set a threshold n_c to prevent the generated graph $\mathcal{G}'$ from having too many tables. After merging, Divo tests if the number of tables $|\mathcal{G}'|$ goes beyond $|\mathcal{G}_1|$, and invokes reduction to randomly erase tables from $|\mathcal{G}_2|$ if so. This strategy preserves the whole structure of $\mathcal{G}_1$ and maintains fidelity. The time complexity of combination is $O(|V_1| + |E_1| + |V_2| + |E_2|)$.

Weighted template evolution. Divo iteratively samples existing templates with a weighted strategy, which not only prevents specific join graphs from being preferred and ensures diverse queries, but also protects fidelity to the original workload. In each iteration, Divo uses reduction or combination to evolve join graphs based on templates sampled from the workload $\mathcal{W}$ with weights. Each generated join graph is assigned a new sampling weight and added to $\mathcal{W}$ for future use. To further attach filter predicates to the templates to form new queries, Divo preserves filter predicates of each source template $\mathcal{G}$ during generation and replaces each filter predicate with another one of the same type from the workload with a probability $p_f = 0.25$. We detail the implementation below.

Given a specific workload $\mathcal{W} = \{\mathcal{G}_1, \mathcal{G}_2, \ldots\}$, we assign an identical initial sample weight $w_{\mathcal{G}}^{(0)} = 1/|\mathcal{W}|$ to all templates $\mathcal{G}$ in the workload. When a template $\mathcal{G}'$ is generated by reduction at step t based on $\mathcal{G}_i$, we mark $\mathcal{G}'$ as a successor of $\mathcal{G}_i$ and split a portion of weight from $w_{\mathcal{G}_i}^{(t-1)}$ to form $w_{\mathcal{G}'}^{(t)}$. The following expressions describe the weight change of $\mathcal{G}_i$ and the new weight assignment:

$$w_{\mathcal{G}'}^{(t)} = w_{\mathcal{G}'}^{(0)} = \gamma \left(w_{\mathcal{G}_i}^{(t-1)} - \lambda w_{\mathcal{G}_i}^{(0)} \right)$$
$$w_{\mathcal{G}_i}^{(t)} = w_{\mathcal{G}_i}^{(t-1)} - w_{\mathcal{G}'}^{(t)}$$

where λ determines the preserved minimal weight for $\mathcal{G}_i$. We use $\lambda = 1/3$ so that $w_{\mathcal{G}_i}^{(t)}$ at any step t preserves at least $1/3$ of the initial weight $w_{\mathcal{G}_i}^{(0)}$. γ sets the ratio of weight split, and we set $\gamma = 1/2$ for reduction in our experiments. Combination uses the same weighting strategy, but the generated template $\mathcal{G}'$ inherits weights from both predecessors $\mathcal{G}_i$ and $\mathcal{G}_j$. We use $\gamma = 1/3$ for join graph combination. Additionally, the common subgraph constraint prefers large join graphs since they are more likely to contain connected maximal common subgraphs. To address the issue, we add a factor to the sample weights and sample templates with the modified weight $\hat{w}_{\mathcal{G}_i}^{(t)} = w_{\mathcal{G}_i}^{(t)} / |\mathcal{G}_i|$ for combination.

We applied the query generator to (1) JOB [12] and Extended JOB [17], (2) DSB [4], and (3) Stack [16], and collected 1,000 queries for each workload. These queries are leveraged as extra knowledge to train Divo and also as a new workload for assessments.

4.3 Property analysis of evolved templates

Divo's template-evolving query generator produces queries that satisfy the properties of diversity, informativeness, and fidelity, with a scalable algorithm of polynomial time complexity. We analyze the properties of the query generator in the following.

Diversity. The diversity of generated queries arises at two scales.

First, join graph operations achieve query-level diversity with a guaranteed vast search space. Consider the largest combined join graph $\tilde{\mathcal{G}} = \bigcup_{i=1}^{n} \mathcal{G}_i$ for the source workload $\mathcal{W}$. For any connected subgraph $\mathcal{G} = (V, E) \subseteq \tilde{\mathcal{G}}$, we can break down $\mathcal{G}$ into a set of one-edge subgraphs $(\{u, v\}, \{e\})$ where $e \in E$. Each edge e must exist in some $\mathcal{G}_j \in \mathcal{W}$, and we can obtain the one-edge subgraph with reduction. Therefore, $\mathcal{G}$ can be constructed by combining these subgraphs. This property enables Divo to generate any query formed with existing join predicates on the underlying schema.

Second, weighted sampling ensures workload-level diversity through fair sampling across query templates. While uniform sampling leads to a preference for a template $\mathcal{G}_i$ when we add $\mathcal{G}'$ evolved from $\mathcal{G}_i$ to $\mathcal{W}$, Divo maintains a constant total weight for all templates evolved from each $\mathcal{G}_i \in \mathcal{W}$ to eliminate the bias.

Informativeness. Divo produces queries informative to LQO training. First, reduction naturally guarantees non-empty results when the original template is non-empty. Consider an execution plan where we join the erased tables last. The subplan corresponding to the reduced template must produce non-empty results to meet the final output. For combination, we verify the evolved queries with real execution. Second, we restrict the evolved join graphs to be *connected* to eliminate Cartesian products.

Fidelity. Divo ensures fidelity with the following two strategies. First, reduction and combination inherently maintain structural fidelity by preserving graph structures from original templates. The fidelity is further reinforced by restricting merged join graphs to share a connected maximal common subgraph, which ensures a coherent relationship between combined templates. Second, Divo prevents frequent sampling of evolved templates distant from the original ones by weight preserving, which exponentially decays sample weights of evolved templates while maintaining a minimum probability for initial ones. The weight of a template with n times of evolution never goes beyond $\gamma^n(1-\lambda)^n/|\mathcal{W}|$.

Scalability. Divo implements both reduction and combination with $O(|V| + |E|)$ complexity. Moreover, Divo requires no training overhead. These properties enable rapid generation of queries at scale.

We will show an in-depth analysis of the template-evolving query generator through experiments in §6.2.

5 Diversity-Enhanced Two-Phase Training

Divo leverages diverse queries to enhance plan generation and comparison, but learning from a diverse training workload faces challenging stability issues. In this section, we introduce the static experiences that augment Divo with diverse generated queries, explain Divo's two-phase training pipeline, and further propose a diversity-aware probability distribution loss to stabilize RL training.

5.1 One-time static experience collection

In Phase I, Divo collects substantial execution plans from diverse queries to establish the *static experiences*, which are leveraged to enhance the performance of both plan generation and comparison. The static experiences address the stability challenges of learning from a large and diverse training workload.

(a) Divo's shared encoder and independent models for downsteam tasks (b) AlphaJoin's separate predictor and comparator (c) GLO's one-model architecture

Fig. 4. An illustration of the model architecture of Divo and previous methods, AlphaJoin [41] and GLO [3].

Stability issues on large training workloads. Divo employs a value-based RL pipeline for plan generation. For each training query, Divo iteratively selects the most promising subplans $p_1, \ldots, p_T$ with a latency predictor that estimates the minimal observed latency $\hat{T}(p_T)$ for each subplan p_t. Upon generating the complete plan p_T, Divo updates each $\hat{T}(p_t)$ with the observed latency and stores subplans in replay memory $\mathcal{M}$, which is a standard RL technique for reusing observed experiences. However, large and diverse workloads undermine the pipeline's stability.

The core challenge stems from LQOs' cold-start issue with the lack of subplan latency knowledge [2, 16]. Large workloads impede bootstrap efficiency due to increased complexity, preventing the latency predictor from effectively learning to select efficient subplans. While pre-collecting subplans from partitioned small workloads appears to address bootstrap issues, this approach remains fundamentally impractical. Note that the minimal observed latency $\hat{T}(p)$ for each subplan p is iteratively updated to inform Divo with the best observed plan generation strategy. The value of $\hat{T}(p)$ remains uncertain until exhaustive exploration of p is completed. Therefore, training with pre-collected subplans impedes crucial updates for observed latencies, informing Divo with outdated plan generation strategies that inevitably degrade performance.

Stable training with static experiences. Divo collects *complete* plans and their latencies from diverse generated queries to establish static experiences $\mathcal{S}$, where '*static*' indicates the stable latency values of complete plans that require no updates throughout training. The static experiences $\mathcal{S}$ complement the replay memory $\mathcal{M}$ with diverse pre-collected plans and improve Divo through two benefits. First, unlike subplans that require consecutive updates of the minimal observed latency, complete plans possess deterministic latency values and retain validity without update, preventing contamination from outdated generation strategies. Second, pre-collected plans enable reuse across training configurations without exploration from scratch.

Experience collection. We combine the generated queries and public workloads (JOB, DSB, Extended JOB, and Stack) to establish a large workload $\mathcal{W}_s$ for static experience collection. To obtain complete plans, we temporarily reuse Divo's latency predictor to generate plans through RL exploration. The collected plans ultimately form the static experiences. To alleviate the difficulty of sufficient exploration, Divo splits $\mathcal{W}_s$ into subsets of 100 queries, denoted as $\mathcal{W}_s^{(1)}, \mathcal{W}_s^{(2)}, \ldots, \mathcal{W}_s^{(N_w)}$, and collect plans separately on each subset. For each query q in a subset $\mathcal{W}_s^{(i)}$, Divo establishes a complete plan p, sends p to PostgreSQL to obtain the execution time $T(p)$, and adds p to experiences $\mathcal{S}^{(i)}$. We repeat this process and collect abundant plans to form the static experiences $\mathcal{S} = \mathcal{S}^{(1)} \cup \ldots \cup \mathcal{S}^{(N_w)}$, which contain 132,000 plans in our experiments.

5.2 RL training enhanced by complete plan comparison

In Phase II, Divo trains the latency predictor and the plan comparator with RL enhanced by diverse static experiences. Instead of directly augmenting the latency predictor to improve plan generation, Divo trains the plan comparator with static experiences in a supervised manner, eventually benefiting plan generation with a parameter-sharing strategy.

The parameter-sharing model architecture. Figure 4(a) shows the model architecture of Divo,

which enables plan generation and comparison to benefit each other through shared query embeddings. We explain Divo's architecture with the pipelines of two previous LQOs, AlphaJoin [41] and GLO [3]. Figure 4(b) shows AlphaJoin's separate pipelines where the latency predictor and the comparator are completely unrelated and cannot improve each other. GLO in Fig. 4(c) instead uses the same model to handle the two different tasks. With the latency predictor directly reused for plan comparison, the plan comparator struggles to recognize poor plans from failed plan generation. In contrast, Divo's latency predictor and comparator share common table embeddings $F_{\text{enc}}(q)$ from the encoder, but use task-specific parameters to allow both *mutual* enhancement and *independent* decisions for plan generation and comparison.

Advantages of parameter-sharing. The parameter-sharing architecture enables improving plan generation by training the plan comparator with complete plans in static experiences while avoiding possibly outdated plan generation strategies. We explain the advantage by analyzing the alternative design of pre-collecting abundant subplans and augmenting the latency predictor directly, which fails because outdated strategies derived from pre-collected subplans cannot be refined. For example, PostgreSQL suggests a sub-optimal first step '$k \bowtie mk$' for JOB 17b and results in 4.2s latency, which likely persists in the static observed latencies. Instead, a superior choice '$ci \bowtie n$' of 2.5s may be recorded as 5.0s if followed by poor subsequent joins, *e.g.*, '$(ci \bowtie n) \bowtie mc$', and be permanently discouraged as the observed latency cannot be updated through exploration. In contrast, training the comparator focuses on the persistently valid final latency and circumvents learning from invalidated strategies. We will give a comprehensive study of training methodologies through ablation studies in §6.3.

5.3 Diversity-aware stable RL training

When training on queries from diverse sources, LQO encounters multiple challenging issues that severely affect model stability. Divo stabilizes RL training with a diversity-aware loss function, which complements latency prediction with an auxiliary task of latency probability distribution.

Obstacles from diverse training queries. Most generator-like LQOs [3, 17, 37] are trained by MSE loss, which diminishes the distance between the prediction and the minimal observed latency and exhibits two key drawbacks.

First, MSE loss introduces scale-dependent bias when addressing various latency ranges across different templates. For example, the latencies of JOB [12] 6c's plans span from 20 to 400 ms, while DSB [4] q100 takes more than 2,000 ms. MSE overweights high-magnitude errors and inadequately distinguishes plans with smaller latencies, intrinsically neglecting optimization for certain queries.

Second, MSE loss cannot resolve similar inputs with distinct latencies. For example, JOB 8c and 8d differ only in the predicate values '*writer*' and '*costume designer*' for column '*rt.role*', which query different data but share identical selectivities. Yet '$ci \bowtie rt$' yields 9× more data in 8c due to the subtle difference, doubling the latency compared with 8d. The example demonstrates that subtly different queries with similar or identical features may produce different latencies and possibly distinct optimal plans. MSE loss inherently fails to handle inconsistencies and leads to instability.

Learning plan generation with diversity-aware loss. Divo proposes the diversity-aware loss to address the issues by complementing MSE with a KL divergence loss for latency predictor training. With the KL divergence loss, Divo fits the minimal observed latency $\hat{T}(p)$ with probabilities of multiple class labels that represent different ranges of latency. The probabilities form a distribution of the latency. For example, a predicted probability 0.6 for the class label of the range $[10^3, 10^{3.25}]$ indicates that $\hat{T}(p)$ has a 60% chance between 10^3 ms and $10^{3.25}$ ms.

Learning to predict probability distribution enhances Divo through three key advantages. First, probability distributions provide scale invariance by transforming latency ranges into class labels

with uniform weights. Second, class labels possess robustness to inconsistencies with natural representations of different latencies. Third, recent progress in deep RL [7] validates the benefit that training value models with probability distributions yields more informative hidden states and higher performance.

Implementation. We model the minimal latency $T_{\min}$ as log-normally distributed since the execution cost is approximately proportional to predicate selectivities. For a subplan p with observed latency $\hat{T}(p)$, we use normal distribution $\mathcal{N}(\log_{10}\hat{T}(p), \sigma^2)$ to describe $\log_{10} T_{\min}(p)$. To obtain a reasonable approximation of σ, we collected the average variance of PostgreSQL's logarithmic latency on DSB template's queries and got $\sigma_{\text{DSB}} \approx 0.121$. Therefore, we set $\sigma = 1/8$ in our experiments. We discretize the latency values $\log_{10} T_{\min}$ into $k_{\text{cls}} = 24$ classes. Since all queries in our experiments can be executed within 10^6 ms, we assume that $\log_{10} T_{\min}$ is within $[0, 6]$ and thus the length of each range is $\tau = 0.25$. For example, when $\hat{T}$ is 1031 ms, the class representing the range $[3.0, 3.25]$ should have the largest probability. We employ an extra MLP tail to predict the probability distribution (denoted as F_{prob}) and use the KL divergence as an objective function apart from MSE to handle the diversity and inconsistency:

$$L_{\text{MSE}} = \frac{1}{|B_{\text{lat}}|} \sum_{p \in B_{\text{lat}}} \left(F_{\text{lat}}(p) - \hat{T}(p) \right)^2$$

$$L_{\text{KL}} = \frac{1}{|B_{\text{lat}}|} \sum_{p \in B_{\text{lat}}} \sum_{i=1}^{k_{\text{cls}}} F_{\text{prob}}^{(i)}(p) \ln \frac{F_{\text{prob}}^{(i)}(p)}{\Phi_i(\log_{10}\hat{T}, \sigma^2)}$$

$$L_{\text{lat}} = L_{\text{MSE}} + L_{\text{KL}}$$

where $F_{\text{prob}}^{(i)}(p)$ is the i-th range's probability, B_{lat} is a batch sampled from the RL replay memory $\mathcal{M}$, and Φ_i represents the cumulative probability of normal distribution in the i-th range. We combine the KL divergence loss with the MSE loss to exploit both advantages, with the former suppressing instabilities and the latter enabling direct prediction of latencies.

Enhanced plan comparison via relative latency distributions. We enhance plan comparison with the diversity-aware KL divergence loss, which enables prediction of relative latency distribution. Divo's comparator predicts the probability distribution of the relative latency $T(p)/T(p_b)$ between generated plan p and the default plan p_b from a traditional optimizer (PostgreSQL's optimizer in our experiments), facilitating fine-grained comparison over binary classification like Lero [46]. For each input plan p, the comparator uses an MLP M_{cmp} to predict the distribution of $T(p)$, denoted as $M_{\text{cmp}}(p)$ below. Divo obtains the relative latency distribution as follows, shown for the case of $T(p) \geq T(p_b)$.

$$\begin{aligned}\Pr\left(\frac{T(p)}{T(p_b)} = 10^{i\tau}\right) &= \sum_{j=i}^{k_{\text{cls}}} \Pr(T(p) = 10^{j\tau}) \cdot \Pr(T(p_b) = 10^{(j-i)\tau}) \\ &= \sum_{j=i}^{k_{\text{cls}}} M_{\text{cmp}}^{(j)}(p) \cdot M_{\text{cmp}}^{(j-i)}(p_b)\end{aligned}$$

We estimate the relative latency with probability predictions by calculating the mathematical expectation, which precisely reveals the performance difference between p and p_b. Note that the result has an exponentially growing coefficient $10^{i\tau}$. To avoid explosive growth, we restrict the

range of i with a parameter $k_{\text{exp}} = 3$.

$$\mathbb{E}\left[\frac{T(p)}{T(p_b)}\right] = \sum_{i=-k_{\text{exp}}}^{k_{\text{exp}}} 10^{i\tau} \Pr\left(\frac{T(p)}{T(p_b)} = 10^{i\tau}\right)$$

$$F_{\text{cmp}}(p, p_b) = \log_{10} \mathbb{E}\left[\frac{T(p)}{T(p_b)}\right]$$

We train the plan comparator with KL divergence loss.

$$L_{\text{cmp}} = \frac{1}{|B_{\text{cmp}}|} \sum_{p \in B_{\text{cmp}}} \sum_{i=1}^{k_{\text{cls}}} M_{\text{cmp}}^{(i)}(p) \ln \frac{M_{\text{cmp}}^{(i)}(p)}{\Phi_i(\log_{10} \hat{T}, \sigma^2)}$$

5.4 The overall training algorithm

We show the training pipeline in Algorithm 1 and clarify our design.

Phase I. Divo collects plans individually for each subset $\mathcal{W}_s^{(i)}$ by repeating a plan generation process: selects a query $q \in \mathcal{W}_s^{(i)}$ and generates a plan p (line 6), executes p and records the execution latency $T(p)$ (line 7-8), and collects p and $T(p)$ into static experiences $\mathcal{S}^{(i)}$ (line 9). The process is repeated until the number of plans reaches a threshold N_{thres}, which we set to 4,000 in experiments. Divo explores various plans with the latency predictor F_{lat}, which is reset after completing the collection process on each subset.

Phase II. In each iteration, Divo processes through the following steps: receives a query q from the training workload $\mathcal{W}$ (line 17), iteratively selects subplans $p_0, p_1, \ldots, p_T$ for q to form the complete plan p_T (line 18), and collects the subplans into RL replay memory $\mathcal{M}$ (line 19). After collection, Divo samples a batch $B_{\text{lat}} \subset \mathcal{M}$ to calculate the latency predictor's loss L_{lat}, and a batch $B_{\text{cmp}} \subset \mathcal{M} \cup \mathcal{S}$ for the plan comparator's loss L_{cmp} (line 22-23). Divo repeats this process to iteratively improve plan generation and comparison.

6 Experiments

We test Divo and existing LQOs on multiple training and testing workload splits to demonstrate the performance. In this section, we first detail the experimental setup, then demonstrate Divo's superior performance over alternative LQOs, and finally validate the contribution of components through ablation studies. Our source code is available at https://github.com/TianyiChen0316/Divo.

6.1 Experimental setup

Workloads and splits. We use the following 4 public workloads and the corresponding underlying data in our experiments.

- **Join Order Benchmark (JOB)** [12] consists of 33 templates and 113 queries. JOB is the most challenging workload with specifically designed filter predicates that enable different optimal plans for queries of the same template.
- **DSB** [4] consists of 15 templates along with a query and data generator. DSB improves TPC-DS [22] with more challenging data distribution and expressive templates. We use a scale factor of 4 and 10 templates without subqueries (unsupported by most LQOs). We generate 5 queries for each template.
- **Extended JOB** [17] consists of 12 templates and 24 queries, which have 12 new join graphs that complement JOB templates.

Algorithm 1 The training process of Divo

Require: diverse query workload $\mathcal{W}_s$, RL training workload $\mathcal{W}$
Ensure: static experiences $\mathcal{S}$, components $(F_{\text{enc}}, F_{\text{lat}}, F_{\text{cmp}})$

Initialize the encoder F_{enc}, the latency predictor F_{lat}, and the comparator F_{cmp}
Split $\mathcal{W}_s$ into $\mathcal{W}_s^{(1)}, \mathcal{W}_s^{(2)}, \ldots, \mathcal{W}_s^{(N_w)}$
for i **in** $1, \ldots, N_w$ **do** ▷ *Static experience collection* (§5.1)
    set $\mathcal{S}^{(i)} \leftarrow \varnothing$
    while $|\mathcal{S}^{(i)}| < N_{\text{thres}}$ **do**
        Generate a plan p for a query $q \in \mathcal{W}_s^{(i)}$
        Execute p to obtain execution latency $T(p)$
        Update observed latency $\hat{T}(p) \leftarrow T(p)$
        Update $\mathcal{S}^{(i)} \leftarrow \mathcal{S}^{(i)} \cup \{p\}$
        Calculate loss L_{lat} with B_{lat} sampled from $\mathcal{S}^{(i)}$
        Update encoder F_{enc} and latency predictor F_{lat} with L_{lat}
    end while
    Update $\mathcal{S} \leftarrow \mathcal{S} \cup \mathcal{S}^{(i)}$
    Reset the encoder F_{enc} and the latency predictor F_{lat}
end for
for epoch **in** $1, \ldots, N_{\text{epoch}}$ **do**
    for $q \in \mathcal{W}$ **do** ▷ *RL with shared parameters* (§5.2)
        Generate subplans $p_0, p_1, \ldots, p_T$ for q
        Update $\mathcal{M} \leftarrow \mathcal{M} \cup \{p_0, p_1, \ldots, p_T\}$
        Execute p to obtain execution latency $T(p)$
        For all p_i, update $\hat{T}(p_i) \leftarrow \min\left\{\hat{T}(p_i), T(p)\right\}$
        Sample B_{lat} from $\mathcal{M}$ and B_{cmp} from $\mathcal{M}$ and $\mathcal{S}$ ▷ *Diversity-aware loss function* (§5.3)
        Calculate loss L_{lat} with B_{lat} and L_{cmp} with B_{cmp}
        Update encoder F_{enc} and latency predictor F_{lat} with L_{lat}
        Update encoder F_{enc} and comparator F_{cmp} with L_{cmp}
    end for
end for

- **Stack** [16] consists of 16 templates and thousands of queries. We remove templates 9 and 10 that contain subqueries and randomly select 5 queries for each template in our experiments.

In addition, we use the template-evolving query generator to generate 3,000 queries, where 1,000 are based on JOB and Extended JOB, 1,000 for DSB, and 1,000 for Stack. The generated queries contain distinct join graphs from public workloads. We randomly pick 100 from the generated JOB queries to form an extra query workload, denoted as *Gen* below.

We combine the previously mentioned workloads to form workload splits and make a comprehensive test of Divo's performance. We use the following three *challenging* workload splits in which test set templates are unseen in the training workloads.

- **Mixed**. We pick a half of the templates from JOB, DSB, Extended JOB, and Stack, as the testing workload, and use the rest as the training workload. This workload split assesses Divo's capability to learn from a diverse workload and grasp the knowledge to reach a reasonable performance on different query templates.

- **Gen-only**. We use Gen as the testing workload and mix all public workloads to form the training workload. The Gen-only workload split further assesses Divo's ability on a wide range of unseen join graphs learning from all available public workloads.
- **Stack-only**. We use Stack for testing and other public workloads for training. The most challenging Stack-only workload split completely removes Stack from training and evaluates (1) the performance without adequate training set diversity, and (2) the stability on a completely unseen workload.

In addition to the three challenging workload settings, we use two simple workload splits established with only JOB queries to perform a comprehensive study of Divo and existing LQOs in workload settings of various difficulty.

- **JOB-Balsa** [37]: a simplified workload split from Balsa where all test set templates are available in the training workload.
- **JOB-Half**: a part of Mixed where only JOB queries are used and test set templates are unseen during training.

Metrics. We use the workload speedup relative to PostgreSQL's optimizer as the main metric, along with the geometric mean relative latency (GMRL) [40] for a comprehensive evaluation. For each workload in experiments, we collect the total latency of each LQO's plans (denoted as L) and PostgreSQL's (L_b). The speedup is then calculated as L_b/L. While the speedup demonstrates *overall* performance, GMRL assesses the performance from a distinct aspect by focusing on the *quantity* of benefits or flaws in the generated plans. For instance, different plans with the same relative latency of 0.5 are considered identical for GMRL's calculation, but the latency can be either 10 ms or 1,000 ms, which affects the speedup differently.

We run Divo on a server of Ubuntu 22.04 with 2 Intel Xeon Gold 5318Y CPUs and 256GB of memory and train Divo with an NVIDIA RTX A5000 GPU. We use a hidden size of 512 for Graphormer and 1024 for QueryFormer. The models are optimized using AdamW with 10^{-2} weight decay and a learning rate of 3×10^{-5}. We train Divo for 60 epochs and take the performance at the final epoch for evaluation. For the compared LQOs, we adhere to the original implementations as specified in their documentation.

Compared methods. We compare Divo with PostgreSQL's built-in optimizer and the following LQOs.

- **Balsa** [37]: a *generator-like* optimizer that bootstraps from a cost simulator and enhances the stability by safe exploration.
- **HybridQO** [39]: a *generator-like* optimizer that generates only the first steps of a plan and completes the plans by sending them to PostgreSQL's optimizer.
- **LOGER** [2]: a *generator-like* optimizer that stabilizes plan generation with reward weighting loss.
- **GLO** [3]: a *generator-like* optimizer aiming to handle unseen tables and workloads. GLO uses a comparator loss to enable complete reuse of the latency predictor to make comparisons.
- **Lero** [46]: a *steerer-like* optimizer that obtains plans by modifying PostgreSQL's cost estimations. Lero pairwisely compares the obtained plans with binary labels from a comparator model.
- **FASTgres** [34]: a *steerer-like* optimizer that learns a hint predictor for each specific template.
- **Eraser** [33]: a *plan comparator* designed for seamless integration with existing LQOs to erase poor plans.
- **LIMAO** [43]: a recently proposed *extension* that integrates existing LQOs with task modules to handle dynamic workload shift. We compare Divo with LIMAO's open-source implementation on Balsa (denoted as LIMAO-Balsa) on dynamic workload settings, omitting the tests on static settings (covered by Balsa).

Table 1. Performance of Divo and compared LQOs on testing workloads of different workload splits.

Competitor			Generator-like LQOs					Steerer-like LQOs		
			Divo	Balsa	HybridQO	LOGER	GLO	Lero	FASTgres	Divo-Eraser
Speedup (↑)	Train	Mixed	**2.03**	0.12	0.05	1.97	2.01	0.83	1.18	0.52
		Gen-only	**2.06**	0.16	0.06	1.08	1.47	0.62	1.47	0.64
		Stack-only	**2.00**	0.59	0.09	1.57	1.06	0.90	1.59	1.27
	Test	JOB-Balsa	1.63	1.82	1.31	1.70	1.81	1.17	**1.83**	1.18
		JOB-Half	1.60	1.73	1.03	1.49	**1.79**	0.87	1.52	1.01
		Mixed	**1.33**	0.06	0.05	0.05	0.37	0.15	1.00	1.00
		Gen-only	**1.28**	0.22	0.08	0.55	0.18	0.67	1.00	1.00
		Stack-only	1.00	0.01	0.01	0.01	0.94	**1.04**	1.00	1.00
		JOB-Balsa	1.57	1.33	1.15	**1.76**	1.76	1.02	1.52	1.21
		JOB-Half	**1.17**	0.35	0.82	0.86	1.10	0.95	1.00	1.00
GMRL (↓)	Train	Mixed	0.65	8.29	10.1	**0.56**	0.71	1.13	0.58	1.09
		Gen-only	**0.67**	8.56	10.1	0.80	0.84	1.07	0.72	1.01
		Stack-only	0.64	3.54	7.68	**0.11**	1.17	1.02	0.60	0.95
	Test	JOB-Balsa	0.62	0.73	0.97	**0.45**	0.75	0.83	0.69	0.86
		JOB-Half	0.61	0.87	0.92	**0.43**	0.65	1.07	0.58	0.99
		Mixed	**0.92**	16.4	16.1	2.77	1.31	1.19	1.00	1.00
		Gen-only	**0.82**	5.96	4.44	1.00	1.43	0.99	1.00	1.00
		Stack-only	**1.00**	169	32.7	22.2	1.03	1.01	**1.00**	**1.00**
		JOB-Balsa	**0.76**	1.31	1.06	1.01	0.81	0.96	0.92	0.87
		JOB-Half	**0.92**	2.11	1.00	1.14	1.07	1.10	1.00	1.00

Table 2. Epochs, data collection overhead, average training time, and average inference time of compared methods.

Competitor	Divo (ours)	Balsa	HybridQO	LOGER	GLO	Lero	FASTgres
Epochs	60	60	60	60	60	60	N/A
Dataset collection (min)	9416	N/A	N/A	N/A	N/A	N/A	**367**
Time per epoch (min)	18.1	187	172	59.7	24.7	90.8	**5.31**
Inference time (s)	0.91	0.17	0.14	1.16	0.86	3.96	**0.11**

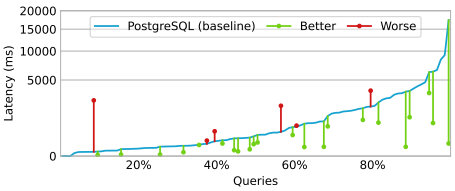


Fig. 5. Divo's performance on each query (sorted by execution latency) in Gen-only testing workload.

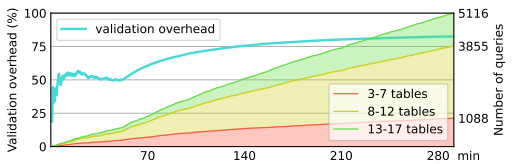


Fig. 6. The percentage of validation overhead and the number of join graphs of different sizes over time.

6.2 Performance evaluation

Experimental design. We design several experiments to evaluate Divo's full pipeline. We assess the speed and join graph diversity of the template-evolving query generator, the plan execution time

on different workload splits compared with existing LQOs, and the training and inference overhead. Furthermore, we perform an in-depth study of Divo's capability under more challenging workload configurations. We assess the impact of join graph size on Divo and verify Divo's ability to learn from join graphs of different complexities. In addition, we run Divo on a dynamic environment where the training and testing workload changes over time to verify Divo's adaptability to workload shift.

Diversity and runtime analysis of query generation. We evaluate the template-evolving query generator on JOB, producing 10,000 new queries with a maximal graph size of 17 to match JOB's complexity. Results confirm Divo achieves high join graph diversity with low overhead. As shown in Fig. 6, Divo generates 5,116 distinct join graphs in 4.86 hours with diversity increasing steadily alongside roughly linear time growth. Moreover, the distribution of join graph complexity remains stable throughout generation. The validation for non-empty results takes 4.01 hours (82.5% of the total runtime) and finds 89 empty queries, which are reduced by the validity guarantee from join graph reduction. As Divo limits the validation to join graph combination only, this guarantee significantly contributes to overall efficiency.

Overall performance comparison. Table 1 summarizes the speed-up and GMRL for Divo and competitors. Leveraging diverse knowledge from both static experiences and training workloads, Divo consistently outperforms compared LQOs across workload splits. Compared with the best generator-like competitor, LOGER [2] and GLO [3], Divo achieves 3.6× and 2.3× speedup on Mixed and Gen-only testing workloads, respectively. On the unseen Stack testing workload, Divo conservatively defaults to PostgreSQL's plans when receiving high uncertainty. While not surpassing PostgreSQL, this strategy still exceeds all generator-like competitors, as they did not properly handle the completely unseen Stack queries. We further investigate the relative latency of each query in the Gen-only testing workload in Fig. 5 to comprehensively analyze Divo's performance. The results confirm that Divo effectively avoids poor plan selection, and the occasional occurrence of flawed plans does not diminish the significant advantage on slow queries.

Our results validate the characteristic trade-offs between LQO architectures. Generator-like LQOs offer the potential of higher performance through vast search spaces, while steerer-like LQOs benefit from the robustness of the underlying traditional optimizer. LOGER and GLO exhibit strong training workload performance but fail to behave properly on testing workloads. GLO partially erases poor plans with a fallback strategy but still performs worse than PostgreSQL. Steerer-like methods show greater stability but still cannot surpass PostgreSQL on diverse unseen templates. Lero [46] demonstrates instability on Mixed and Gen-only, while FASTgres [34] cannot utilize knowledge from different templates and always defaults to PostgreSQL's plan on testing workloads.

Divo outperforms existing methods by effectively combining the high potential of generator-like LQOs, the stability of steerer-like LQOs, and the knowledge from diverse queries. Diverse static experiences enable high performance for plan generation, while the plan comparator eliminates flaws and ensures reliability. This approach requires appropriate handling of diverse queries. We replace Divo's plan comparator with Eraser [33] to demonstrate its performance. Eraser suffers from uncertainty and falls back to PostgreSQL's plan on most testing workloads. Moreover, Eraser exhibits noticeable performance limitations when processing training workloads, demonstrating its limited capability to handle diverse queries. In contrast, Divo's plan comparator learns from diverse experiences, consistently stabilizing and improving Divo's performance.

Performance on simplified workloads. To determine whether performance degradations of the compared LQOs stem from challenging workload settings, we evaluate Divo and the comparators on simplified workloads, JOB-Balsa and JOB-Half. The results in Table 1 show that all competitors recover their performance on JOB-Balsa, where test set templates are contained within the training

Table 3. Testing workload speedup of Divo and LIMAO-Balsa on dynamic workloads.

Workload	IMDB-LIMAO			Mixed-Shift				
	IMDB-1	IMDB-2	**All**	JOB	ExtJOB	DSB	Stack	**All**
LIMAO	2.44	2.66	2.59	0.29	0.56	0.04	0.01	0.07
Divo	**2.49**	2.86	**2.76**	**1.11**	1.47	1.00	**1.07**	**1.17**
– without plan comparison	2.30	**2.92**	2.74	0.69	**1.48**	**1.06**	0.08	0.63

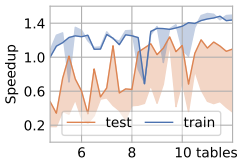


Fig. 7. Speedup change over increasing graph size.

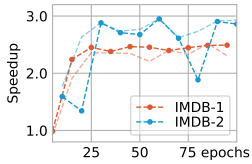


Fig. 8. Speedup on IMDB- LIMAO with dynamic shift.

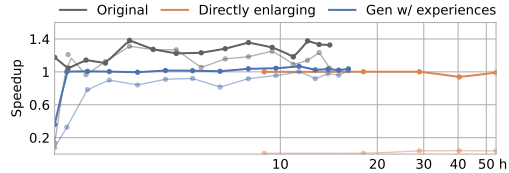


Fig. 9. Speedup of Divo on Mixed testing workload with different design choices of static experiences.

workload. On the more challenging JOB-Half split, most LQOs remain slightly inferior to PostgreSQL. Nonetheless, both generator-like and steerer-like methods show higher speedups on JOB-Half than on diversified workload splits. These results confirm that degradation primarily stems from the inability to handle diverse queries, while unseen test set templates also contribute to the phenomenon.

Performance variation with increasing complexity. We evaluate Divo's capability to leverage queries of varying complexity by training it on a progressively expanding workload and testing on the Gen-only testing workload. The training process begins with queries of 3 to 6 tables for the first 24 epochs, with 20 queries per category. After every 16 epochs, we expand the training workload by 20 additional queries, with a join graph size increased by 1. Thus, Divo is trained with queries of 3 to 7 tables by epoch 40. The process of workload expansion continues until epoch 120, where the size eventually reaches 12.

Figure 7 shows the speedup progression as query complexity increases, where shadings indicate the contribution of plan comparison. Divo demonstrates steady performance improvements on both training and testing workloads, thus confirming its ability to grasp knowledge from queries of different complexities. While the initial average speedup trained on queries of 3 to 6 tables reaches only 0.65×, Divo progressively improves the performance and finally achieves 1.12× on the testing workload. These results also reveal the necessity of diverse training queries to cover a spectrum of complexities and equip LQOs with sufficient knowledge for robust performance on future inputs.

Performance under dynamic workload shifts. To evaluate robustness in realistic scenarios with frequent workload changes, we test Divo under two dynamic settings:

- **IMDB-LIMAO** [43]: LIMAO's workload settings with periodic switches between two IMDB-based splits every 5 epochs. We denote the workloads as IMDB-1 and IMDB-2, which partially overlap with each other.
- **Mixed-Shift**: a variant of Mixed workload split where training and testing workloads periodically rotate among JOB, Extended JOB, DSB, and Stack.

Figure 8 shows Divo's consistent performance improvement on both IMDB-1 and IMDB-2 despite periodic shifts. We show detailed results in Table 3 with a comparison to LIMAO-Balsa. While LIMAO-Balsa exhibits competitive performance on overlapping IMDB queries, it struggles to switch

Table 4. The results of ablation studies. The 'NoCmp' columns show the performance of Divo before plan comparison.

Metric	Name	Mixed				Gen-only				Stack-only			
		Train	NoCmp	**Test**	NoCmp	Train	NoCmp	**Test**	NoCmp	Train	NoCmp	**Test**	NoCmp
Speedup (↑)	Ori	2.03	2.06	**1.33**	1.04	**2.06**	**2.13**	**1.28**	0.30	2.00	1.99	**1.00**	0.02
	NoExp	2.02	2.01	1.12	0.93	2.01	1.98	0.35	0.20	1.96	1.98	0.98	0.65
	HalfExp	**2.06**	2.06	1.21	1.01	1.56	1.54	1.11	0.81	**2.01**	2.01	**1.00**	0.90
	NoGen	2.05	2.05	1.12	1.04	1.58	1.57	1.16	0.54	1.99	**2.01**	**1.00**	0.74
	LatExp	1.68	1.63	1.04	1.01	1.54	1.46	1.04	0.41	1.41	1.36	**1.00**	**1.03**
	NoKL	2.00	**2.07**	1.04	0.97	2.04	2.05	1.26	**1.22**	1.99	2.00	**1.00**	0.96
	NoMSE	1.49	0.13	0.98	0.09	1.07	0.06	1.01	0.12	1.38	0.24	**1.00**	0.01
	Reuse	2.04	2.04	1.00	**1.10**	2.00	1.98	1.17	0.22	1.97	1.96	**1.00**	0.11
GMRL (↓)	Ori	**0.65**	**0.64**	0.92	1.11	**0.67**	**0.65**	**0.82**	1.11	0.64	0.64	**1.00**	3.50
	NoExp	0.68	0.68	0.95	**1.06**	0.70	0.69	1.22	1.35	0.66	0.65	1.03	1.24
	HalfExp	0.66	0.66	**0.91**	1.13	0.70	0.70	0.94	**1.00**	**0.63**	**0.63**	**1.00**	0.99
	NoGen	0.68	0.69	0.92	1.07	0.71	0.70	0.98	1.25	0.65	0.65	**1.00**	1.13
	LatExp	0.84	0.85	0.96	1.06	0.85	0.86	0.93	1.22	0.86	0.89	**1.00**	**1.01**
	NoKL	0.68	0.66	0.96	1.15	0.69	0.68	0.91	1.03	0.64	0.64	**1.00**	1.01
	NoMSE	0.93	4.52	1.01	4.56	0.97	8.27	0.99	6.83	0.93	3.61	**1.00**	28.7
	Reuse	0.67	0.66	0.97	1.08	0.69	0.70	0.98	1.19	0.65	0.65	**1.00**	1.46

among thoroughly different workloads in Mixed-Shift. Instead, Divo's plan comparator effectively filters poor plans and ensures more stable performance. Note that Divo achieves 1.17× test set speedup in total on Mixed-Shift, inferior to 1.33× speedup in static Mixed workload settings shown in Table 1. This result indicates that the inability to access queries from multiple workload sources simultaneously limits Divo's performance under dynamic workload settings.

Training and inference time. Table 2 shows the data collection overhead, average training time, and inference time for Divo and other LQOs. All competitors were trained for 60 epochs for a fair comparison. Balsa and HybridQO exhibit high per-epoch training time, while Lero's full retraining from scratch in each epoch increases its overhead. Divo trains faster than GLO since static experiences can also be leveraged as a lookup table of latencies and reduce redundant executions. Although Divo requires a collection phase of about 157 hours, collected experiences are reusable across different configurations and incur no extra training overhead.

For inference, Divo requires more time than Balsa, HybridQO, and FASTgres, as the latter benefit from simple model architectures. However, Divo's inference time is competitive with LOGER and GLO, and substantially lower than Lero. Given that none of the existing LQOs produce efficient plans over diverse workloads, we prioritize the pivotal task of performance improvement and leave inference acceleration to our future work.

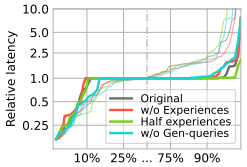
6.3 Ablation study

To illustrate the effectiveness of different components, we evaluate Divo on the three workload splits with different modifications and analyze the performance. We demonstrate all results in Table 4, explain the experiment names in Table 5, and detail the performance in Fig. 9-12.

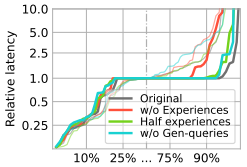
Design choices of training with static experiences. We test Divo with the following modifications: (1) directly enlarging the training workload with generated queries, (2) augmenting the

Table 5. Experiment names and corresponding descriptions for ablation studies.

Experiment name	Description
Ori	The original Divo implementation.
NoExp	Training Divo without static experiences.
HalfExp	Training Divo with half of the experiences.
NoGen	Training Divo without experiences from generated queries.
LatExp	Training the latency predictor with static experiences.
NoKL	Training Divo without KL divergence loss.
NoMSE	Training Divo without MSE loss.
Reuse	Reusing the latency predictor to make plan comparison.



(a) Mixed



(b) Gen-only

Fig. 10. Relative latency of generated plans on different testing workloads by percentage.

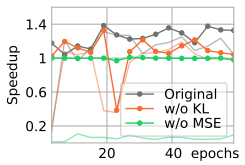


Fig. 11. Speedup without KL divergence loss or MSE loss on Mixed.

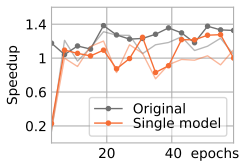


Fig. 12. Speedup with latency predictor reused for comparison on Mixed.

generator with collected *subplans*, (3) training without static experiences, (4) training with half the static experiences, and (5) excluding generated queries from static experiences.

Figure 9 demonstrates the speedup of (1) and (2) over time on Mixed workload split. Solid and transparent lines denote the performance before and after plan comparison, respectively. The distance between contiguous points on the curves represents 4 epochs, which is the validation interval in our experiments. The results reveal that directly expanding the training workload not only substantially increases the training overhead but also yields disastrously poor plans, validating our claim that existing training methods cannot work properly on large diverse workloads. We omit the detailed performance of (1) from Table 4 as training for 60 epochs takes too much time. The naive design choice (2) of training with collected subplans performs better than (1) but remains inferior to PostgreSQL. This failure stems from learning outdated plan generation strategies of static minimal observed latencies, which become invalidated during RL exploration.

We examine design choice (3) to verify the benefit of training with collected static experiences. Figure 10 demonstrates the latency of generated plans relative to PostgreSQL on Mixed and Gen-only testing workload by percentage. From the red curve in Fig. 10(b), we can see that Divo fails to reduce poor plans on generated queries without static experiences, resulting in significant degradation. We further explain the importance of static experiences by using only half of the experiences in design choice (4). From the results in Table 4, we can see a gradual improvement of speedup from 1.12 to 1.21 and finally 1.33 on the Mixed testing workload when we increase the use of static experiences. A similar improvement can be seen on Gen-only (0.25, 1.11, and 1.28). The results exactly verify the effectiveness of Divo's two-phase training pipeline.

Design choice (5) removes generated queries from static experiences to isolate the contribution of the template-evolving query generator. Results in Table 4 demonstrate substantial speedup gains from 0.92 to 1.33 on Mixed testing workload and 0.98 to 1.28 on Gen-only, validating the critical

role of generated queries. By producing numerous synthetic queries with join graphs distinct from public workloads, the template-evolving query generator furnishes Divo with abundant, diverse knowledge.

Design choices of the diversity-aware loss. We test the performance of Divo with the following modifications: (1) removing the KL divergence loss, and (2) removing the MSE loss. For the latter, Divo replaces latency prediction by calculating the mathematical expectation of latency with probability distribution.

We show the test set speedup of disabling KL divergence or MSE loss over time in Fig. 11, where transparent curves represent the speedup of generated plans without comparison. The 'w/o KL' variant is almost always inferior to the original, indicating the KL divergence loss's apparent benefit in training stability and performance. Moreover, we observe a surprising finding that Divo behaves improperly without explicit value prediction. With MSE loss removed, Divo consistently generates poor plans and fails to outperform PostgreSQL even with plan comparison. The results confirm the necessity of integrating both losses, where the KL divergence ensures robustness and the MSE provides an explicit criterion for plan generation.

An alternative to the model architecture. To examine the impact of different model architectures as shown in Fig. 4 on Divo, we reuse the latency predictor to perform plan comparison with a single model as an alternative that resembles GLO [3]. We can see from Table 4 that the speedup on Mixed and Gen-only decreases to different extents. Figure 12 shows the test set speedup over time. With a single model handling both latency prediction and comparison, Divo generates poorer plans and often fails to erase them, resulting in a fluctuating performance. As the completely shared model indicates inherent dependence between the latency predictor and the comparator, plan generation and comparison are likely to fail together, revealing the necessity of independent model parameters.

In conclusion, Divo's performance benefits from all the proposed components, including queries from the template-evolving query generator, the two-phase collecting and training strategy, the parameter-sharing model architecture, and the diversity-aware loss function. With the delicate design of the whole training pipeline, Divo performs better than PostgreSQL and all compared LQOs.

7 Conclusion

We propose Divo, an LQO built on diverse workloads. First, we establish a template-evolving query generator that performs join graph operations on existing templates to produce diverse, informative queries with fidelity at scale. Second, we design a two-phase model training pipeline to equip Divo with diverse static experiences through parameter-sharing RL training. Third, we introduce a diversity-aware probability distribution loss, which ensures training stability on diverse queries and enables fine-grained plan comparisons. Experiments show that Divo consistently outperforms PostgreSQL's built-in optimizer and existing LQOs across three challenging workload splits. Moreover, the proposed components evidently contribute to Divo's superior performance.

Divo is still heading towards being practical. In the future, we will investigate leveraging more sophisticated techniques to accelerate inference, integrating selections of physical join and scan operators to seek higher performance, and adapting Divo to the context of distributed database systems [8].

Acknowledgments

This work is supported by NSFC (No. 62272008) and ZTE-PKU joint program.

References

[1] Nicolas Bruno, Surajit Chaudhuri, and Dilys Thomas. 2006. Generating queries with cardinality constraints for dbms testing. IEEE Transactions on Knowledge and Data Engineering 18, 12 (2006), 1721–1725.

[2] Tianyi Chen, Jun Gao, Hedui Chen, and Yaofeng Tu. 2023. LOGER: A Learned Optimizer Towards Generating Efficient and Robust Query Execution Plans. Proceedings of the VLDB Endowment 16, 7 (2023), 1777–1789.

[3] Tianyi Chen, Jun Gao, Yaofeng Tu, and Mo Xu. 2024. GLO: Towards Generalized Learned Query Optimization. In 2024 IEEE 40th International Conference on Data Engineering (ICDE). IEEE, 4843–4855.

[4] Bailu Ding, Surajit Chaudhuri, Johannes Gehrke, and Vivek Narasayya. 2021. DSB: A decision support benchmark for workload-driven and traditional database systems. Proceedings of the VLDB Endowment 14, 13 (2021), 3376–3388.

[5] Pierluca D'Oro, Max Schwarzer, Evgenii Nikishin, Pierre-Luc Bacon, Marc G Bellemare, and Aaron Courville. 2022. Sample-efficient reinforcement learning by breaking the replay ratio barrier. In Deep Reinforcement Learning Workshop NeurIPS 2022.

[6] Alexandra Duminil, Sio-Song Ieng, and Dominique Gruyer. 2024. Assessing fidelity in synthetic datasets: A multi-criteria combination methodology. In 2024 27th International Conference on Information Fusion (FUSION). IEEE, 1–8.

[7] Jesse Farebrother, Jordi Orbay, Quan Vuong, Adrien Ali Taïga, Yevgen Chebotar, Ted Xiao, Alex Irpan, Sergey Levine, Pablo Samuel Castro, Aleksandra Faust, et al. 2024. Stop regressing: training value functions via classification for scalable deep RL. In Proceedings of the 41st International Conference on Machine Learning. 13049–13071.

[8] Jun Gao, Yinjun Han, Yang Lin, Hao Miao, and Mo Xu. 2024. Learned Distributed Query Optimizer: Architecture and Challenges. ZTE Communications 22, 2 (2024), 49–54.

[9] Yuxing Han, Ziniu Wu, Peizhi Wu, Rong Zhu, Jingyi Yang, Liang Wei Tan, Kai Zeng, Gao Cong, Yanzhao Qin, Andreas Pfadler, et al. 2021. Cardinality estimation in DBMS: a comprehensive benchmark evaluation. Proceedings of the VLDB Endowment 15, 4 (2021), 752–765.

[10] Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, et al. 2022. Training compute-optimal large language models. In Proceedings of the 36th International Conference on Neural Information Processing Systems. 30016–30030.

[11] Sanjay Krishnan, Zongheng Yang, Ken Goldberg, Joseph Hellerstein, and Ion Stoica. 2018. Learning to optimize join queries with deep reinforcement learning. arXiv preprint arXiv:1808.03196 (2018).

[12] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2015. How good are query optimizers, really? Proceedings of the VLDB Endowment 9, 3 (2015), 204–215.

[13] Guoliang Li, Xuanhe Zhou, Shifu Li, and Bo Gao. 2019. Qtune: A query-aware database tuning system with deep reinforcement learning. Proceedings of the VLDB Endowment 12, 12 (2019), 2118–2130.

[14] Ruibo Liu, Jerry Wei, Fangyu Liu, Chenglei Si, Yanzhe Zhang, Jinmeng Rao, Steven Zheng, Daiyi Peng, Diyi Yang, Denny Zhou, and Andrew M. Dai. 2024. Best Practices and Lessons Learned on Synthetic Data. In First Conference on Language Modeling.

[15] Xinyue Liu, Xiangnan Kong, Lei Liu, and Kuorong Chiang. 2018. TreeGAN: syntax-aware sequence generation with generative adversarial networks. In 2018 IEEE International Conference on Data Mining (ICDM). IEEE, 1140–1145.

[16] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Nesime Tatbul, Mohammad Alizadeh, and Tim Kraska. 2021. Bao: Making learned query optimization practical. In Proceedings of the 2021 International Conference on Management of Data. 1275–1288.

[17] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. 2019. Neo: a learned query optimizer. Proceedings of the VLDB Endowment 12, 11 (2019).

[18] Ryan Marcus and Olga Papaemmanouil. 2018. Deep reinforcement learning for join order enumeration. In Proceedings of the First International Workshop on Exploiting Artificial Intelligence Techniques for Data Management. 1–4.

[19] Chaitanya Mishra, Nick Koudas, and Calisto Zuzarte. 2008. Generating targeted queries for database testing. In Proceedings of the 2008 ACM SIGMOD international conference on Management of data. 499–510.

[20] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. 2013. Playing Atari with Deep Reinforcement Learning. arXiv preprint arXiv:1312.5602 (2013).

[21] Neha Patki, Roy Wedge, and Kalyan Veeramachaneni. 2016. The Synthetic Data Vault. In 2016 IEEE International Conference on Data Science and Advanced Analytics (DSAA). 399–410. doi:10.1109/DSAA.2016.49

[22] Meikel Poess, Bryan Smith, Lubor Kollar, and Paul Larson. 2002. Tpc-ds, taking decision support benchmarking to the next level. In Proceedings of the 2002 ACM SIGMOD international conference on Management of data. 582–587.

[23] Ermu Qiu, Jun Gao, Yaofeng Tu, and Jingru Yang. 2025. LIFTus: An Adaptive Multi-Aspect Column Representation Learning for Table Union Search. In 2025 IEEE 41st International Conference on Data Engineering (ICDE). IEEE, 2174–2187.

[24] Sonal Sannigrahi, Thiago Fraga-Silva, Youssef Oualil, and Christophe Van Gysel. 2024. Synthetic query generation using large language models for virtual assistants. In Proceedings of the 47th International ACM SIGIR Conference

on Research and Development in Information Retrieval. 2837–2841.
[25] P Griffiths Selinger, Morton M Astrahan, Donald D Chamberlin, Raymond A Lorie, and Thomas G Price. 1989. Access path selection in a relational database management system. In Readings in Artificial Intelligence and Databases. Elsevier, 511–522.
[26] Jiahao Shen, Ke Jiang, and Xiaoyang Tan. 2023. Boundary Data Augmentation for Offline Reinforcement Learning. ZTE Communications 21, 3 (2023), 29–36.
[27] Donald R Slutz. 1998. Massive Stochastic Testing of SQL. In Proceedings of the 24rd International Conference on Very Large Data Bases. 618–622.
[28] Immanuel Trummer, Junxiong Wang, Deepak Maram, Samuel Moseley, Saehan Jo, and Joseph Antonakakis. 2019. Skinnerdb: Regret-bounded query evaluation via reinforcement learning. In Proceedings of the 2019 International Conference on Management of Data. 1153–1170.
[29] Allan Tucker, Zhenchen Wang, Ylenia Rotalinti, and Puja Myles. 2020. Generating high-fidelity synthetic patient data for assessing machine learning healthcare software. NPJ digital medicine 3, 1 (2020), 147.
[30] Veniamin Veselovsky, Manoel Horta Ribeiro, Akhil Arora, Martin Josifoski, Ashton Anderson, and Robert West. 2023. Generating faithful synthetic data with large language models: A case study in computational social science. arXiv preprint arXiv:2305.15041 (2023).
[31] Xu Wang, Sen Wang, Xingxing Liang, Dawei Zhao, Jincai Huang, Xin Xu, Bin Dai, and Qiguang Miao. 2024. Deep Reinforcement Learning: A Survey. IEEE Transactions on Neural Networks and Learning Systems 35, 4 (2024), 5064–5078. doi:10.1109/TNNLS.2022.3207346
[32] Zige Wang, Wanjun Zhong, Yufei Wang, Qi Zhu, Fei Mi, Baojun Wang, Lifeng Shang, Xin Jiang, and Qun Liu. 2023. Data management for large language models: A survey. arXiv e-prints (2023), arXiv–2312.
[33] Lianggui Weng, Rong Zhu, Di Wu, Bolin Ding, Bolong Zheng, and Jingren Zhou. 2024. Eraser: Eliminating performance regression on learned query optimizer. Proceedings of the VLDB Endowment 17, 5 (2024), 926–938.
[34] Lucas Woltmann, Jerome Thiessat, Claudio Hartmann, Dirk Habich, and Wolfgang Lehner. 2023. Fastgres: Making learned query optimizer hinting effective. Proceedings of the VLDB Endowment 16, 11 (2023), 3310–3322.
[35] Chenhao Xu, Chunyu Chen, Jinglin Peng, Jiannan Wang, and Jun Gao. 2025. BQSched: A Non-Intrusive Scheduler for Batch Concurrent Queries via Reinforcement Learning. In 2025 IEEE 41st International Conference on Data Engineering (ICDE). IEEE Computer Society, 183–196.
[36] Suorong Yang, Weikang Xiao, Mengchen Zhang, Suhan Guo, Jian Zhao, and Furao Shen. 2022. Image data augmentation for deep learning: A survey. arXiv preprint arXiv:2204.08610 (2022).
[37] Zongheng Yang, Wei-Lin Chiang, Sifei Luan, Gautam Mittal, Michael Luo, and Ion Stoica. 2022. Balsa: Learning a Query Optimizer Without Expert Demonstrations. In Proceedings of the 2022 International Conference on Management of Data. 931–944.
[38] Chengxuan Ying, Tianle Cai, Shengjie Luo, Shuxin Zheng, Guolin Ke, Di He, Yanming Shen, and Tie-Yan Liu. 2021. Do transformers really perform badly for graph representation? Advances in neural information processing systems 34 (2021), 28877–28888.
[39] Xiang Yu, Chengliang Chai, Guoliang Li, and Jiabin Liu. 2022. Cost-based or learning-based? A hybrid query optimizer for query plan selection. Proceedings of the VLDB Endowment 15, 13 (2022), 3924–3936.
[40] Xiang Yu, Guoliang Li, Chengliang Chai, and Nan Tang. 2020. Reinforcement learning with tree-lstm for join order selection. In 2020 IEEE 36th International Conference on Data Engineering (ICDE). IEEE, 1297–1308.
[41] Ji Zhang, K Zhou, and S Schelter. 2020. AlphaJoin: Join Order Selection à la AlphaGo. PhD@ VLDB 2652 (2020).
[42] Lixi Zhang, Chengliang Chai, Xuanhe Zhou, and Guoliang Li. 2022. Learnedsqlgen: Constraint-aware sql generation using reinforcement learning. In Proceedings of the 2022 International Conference on Management of Data. 945–958.
[43] Qihan Zhang, Shaolin Xie, and Ibrahim Sabek. 2025. LIMAO: A Framework for Lifelong Modular Learned Query Optimization. arXiv preprint arXiv:2507.00188 (2025).
[44] Yue Zhao, Gao Cong, Jiachen Shi, and Chunyan Miao. 2022. QueryFormer: a tree transformer model for query plan representation. Proceedings of the VLDB Endowment 15, 8 (2022), 1658–1670.
[45] Kai Zhong, Luming Sun, Tao Ji, Cuiping Li, and Hong Chen. 2024. FOSS: A Self-Learned Doctor for Query Optimizer. In 2024 IEEE 40th International Conference on Data Engineering (ICDE). IEEE, 4329–4342.
[46] Rong Zhu, Wei Chen, Bolin Ding, Xingguang Chen, Andreas Pfadler, Ziniu Wu, and Jingren Zhou. 2023. Lero: A Learning-to-Rank Query Optimizer. Proceedings of the VLDB Endowment 16, 6 (2023), 1466–1479.

Received July 2025; revised October 2025; accepted November 2025